Web Services and Cloud-Based Systems Assignment 2

Martin Warnaar (10689613) and Bryan te Beek (10690441)

2.1 - Automatically deploying Web Service

Experience with ONE API

We found that the ONE API was quite easy to use, especially the CLI tools. The contextualization of a virtual machine was the toughest part, but making Glassfish work with our .war file isn't really part of the ONE API, but has more to do with the OS and installing / managing packages and the way Glassfish works. The manual pages for the various CLI tools are well written and we used them quite a lot to find out about the available commands and how we could leverage them. Finding extra documentation / tutorials about the ONE API on the web was a lot harder, we could not seem to find many tutorials on how to do more complex things with the ONE API.

Contextualisation of the VM

To make sure our web service is running on the VM when it is started, we used Glassfish' 'asadmin create-service', which creates a startup script for domain1 that contains the application. We added our .war file in the auto deployment folder of Glassfish so that it will get deployed automatically. By using Glassfish the web service is publicly (to the DAS cluster, it's not possible to make an HTTP request from any IP that is external to the DAS cluster) available automatically on port 8080. Next, we created a snapshot using the ONE api on the DAS cluster and we used that to create a new VM instance with the provided pumpkin.one file. This way, the RESTful web service was automatically ran once the new VM was booted up. Glassfish takes rather long to start the domain containing the application to deploy, but after waiting a while it worked as expected.

2.2 - Classifying tweets using Pumpkin workflows

Experience with Pumpkin

We found that Pumpkin was a bit hard to use because it wasn't really verbose in outputting useful errors. Most of the errors that arose while we were experimenting with Pumpkin, were fatal errors in the python scripts that power it. Since we both are not python experts, it was somewhat difficult to find the exact problem that resulted in the fatal errors. Though, most of the errors that occurred were simply because of missing files on the filesystem or a miss configuration of some of the Pumpkin related configuration files. These errors should be easy to catch in the core framework which could help towards creating more verbose error messages so the person leveraging the framework can address the problems effectively.

After we had overcome all the basic errors that involved missing files and misconfigurations, we could see the true power of the Pumpkin framework and were happily surprised by it's effectiveness. It was extremely easy to split the different jobs to different seed nodes and therefore create a small private cloud that leveraged communication using messages to solve a larger problem.

Performance of different seeds combinations

There were three setups in which we ran the classifier. On the first setup we used a single VM that ran the collector, the injector and the filter. On the second setup we used two VM's; one VM that ran both the collector and the injector and another VM that ran only the filter. On the third setup we used three VM's that had the collector, the injector and the filter respectively, splitting the problem between all three of them.

It was hard to get a lot of statistics about the actual performance of the different setups, mainly because we had seen a lot of hiccups while using the DAS4 cluster while we were merely using any of our resources. All three setups required quite a bit of time to run and because of the frequent hiccups we cannot reliably say something about the actual performance difference between the three setups. Though, if we imagine an isolated case where the performance would not be affected by other users, we would most likely see an improvement in performance the more we split the jobs on different VM's.